Computing Hypergraph Homology in Chapel

Jesun Sahariar Firoz*, Louis Jenkins†, Cliff Joslyn*, Brenda Praggastis*, Emilie Purvine*, Mark Raugas*
{jesun.firoz,brenda.praggastis,emilie.purvine,cliff.joslyn,mark.raugas}@pnnl.gov, ljenkin4@ur.rochester.edu

*Pacific Northwest National Laboratory
Seattle, WA, USA.

† University of Rochester
Rochester, NY, USA.

***Abstract*—In this paper, we discuss our experience in implementing homology computation, in particular the Betti number calculations in Chapel hypergraph Library (CHGL). Given a dataset represented as a hypergraph, a Betti number for a particular dimension k indicates how many k-dimensional 'voids' are present in the dataset. Computing the Betti numbers involve various array-centric and linear algebra operations. We demonstrate that implementing these operations in Chapel is both concise and intuitive. In addition, we show that Chapel provides language constructs for implementing parallel and distributed execution of the linear algebra kernels with minimal effort. Syntactically, Chapel provides succinctness of Python, while delivering comparable and better performance than C++-based and Julia-based packages for calculating the Betti numbers respectively.**

***Index Terms*—Chapel, computational topology, hypergraphs**

I. Introduction

In this paper, we discuss the implementation of algorithms representing complex network data as hypergraphs and the computation of their homology groups, in particular computation of the Betti numbers, in Chapel. A hypergraph $H = \langle V, E \rangle$ is a mathematical object, similar to a graph, with vertices V connected by edges E (fig. 1). But hypergraphs are more general than graphs, in that where graph edges connect pairs of vertices, hypergraph edges $e \in E$ may contain any arbitrary number of vertices $e \subseteq V$. In particular, every graph is a 2-uniform hypergraph. Hypergraph metrics also generalize graph metrics, so that network concepts like centrality and clustering coefficients extend in a natural way to hypergraph objects [1].

Moreover, as collections of connected multi-dimensional objects, hypergraphs are *topological* objects, and have measurable topological properties. In topology, an n-simplex is the convex hull of any $n+1$ points in $\mathbb{R}^{n+1}$. For example, a 0-simplex is a single point, a 2-simplex is a filled in triangle etc. In this paper we discuss the computation of the topological invariants of hypergraph networks as measured by homology. These features reveal themselves as "holes" or "voids" of different dimension. For example, the hypergraph $\{\{a,b,c\},\{c,d\},\{b,d\}\}$ is a 2-simplex adjoining an implied 3-clique $\{\{b,c\},\{c,d\},\{b,d\}\}$, and has a 1-dimensional hole inside that open cycle; while the hypergraph $\{\{a,b,c\},\{b,c,d\}\}$ consists of two joined 2-simplices, and so is a 2-dimensional solid with no hole or void. Holes and voids of different dimension may correspond to data that is missing or anomalous in some way. In this paper, we are primarily interested in finding the number of $k-$dimensional holes in a given dataset, namely the Betti number k.

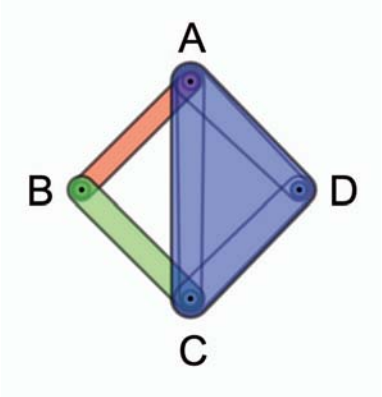

Fig. 1. The **Tri-Loop** hypergraph has three edges colored red, blue and green. Its associated ASC has ten edges, which are outlined in grey.

Mathematically, to calculate the Betti numbers, we first infer all relationships internal to the hyperedges by computing their *abstract simplicial complexes (ASC)*. We identify the collection of subsets of a specific size $k+1$ with a set of generators for a vector space, C_k, over $\mathbb{Z}_2$. We compute *boundary maps*, $\partial_k : C_k \to C_{k-1}$, that associate elements of C_k, which are collections of k dimensional edges, with their $k-1$ dimensional boundaries. The rank, kernel, and image of the corresponding matrices provide the dimensions and bases for the homology groups. We can trace the basis elements back to the data to discover the source of the anomalous behavior in the network. Our implementation uses the Smith Normal Form of the boundary matrices M. From this representation it is possible to read off the rank and compute the desired homology bases using matrix multiplication. We implement our algorithms in the Chapel Hypergraph Library (CHGL) [2], developed to support hypergraph computation at scale.

The paper is organized as follows. We discuss the steps involved in Betti number calculation with an example in details in section II. We describe our implementation of the Betti number calculation in Chapel in section III. We present our experimental results in section IV. Finally we summarize our

2020 Chapel Implementers and Users Workshop (CHIUW 2020). Supported by the US Department of Energy (DOE) Computational Science Graduate Fellowship grant DE-SC0020347, and the High Performance Data Analytics program at the DOE's Pacific Northwest National Laboratory, operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

978-1-7281-7445-7/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPSW50202.2020.00112

conclusion and future work in section V.

II. Algorithm for Computing Betti Numbers

We illustrate the methodology of computing the Betti numbers with the Tri-loop hypergraph. The Tri-Loop hypergraph, H, (Fig. 1) has three edges $\{A, B\}, \{B, C\}$, and $\{A, C, D\}$. The steps for computing the Betti numbers are:

1) Compute the unique abstract simplicial cell complex (ASC) associated with H. Let X be the hypergraph with edges given by the set of all non-empty subset of the edges in H. X is an ASC. An edge of X is called a $k-$cell, where k= dimension of the cell = (number of nodes) - 1. In our example, $X = \{A, B, C, D, AB, AC, AD, BC, CD, ACD\}$.
2) Group all the k-cells together. In our example:
   0-cells: $\{A, B, C, D\}$.
   1-cells: $\{AB, AC, AD, BC, CD\}$.
   2-cells: $\{ACD\}$.
3) Compute the boundary matrices, ∂_k. Each $k-$cell has a boundary that is a union of $(k-1)$-cells. The boundary map is a linear homomorphism (for example, $\partial_1(AB) = A+B$ in **Tri-Loop**). In **Tri-Loop**, the matrix representation for ∂_1 has rows indexed by the ordered set of 0-cells, A, B, C, D, and columns indexed by the ordered set of 1-cells, AB, AC, AD, BC, CD. We include a 1 if there is a matching subsequence between $(k-1)$-cell and k-cell. For example, $\partial_1[1, 1] = 1$, since A is contained in AB.
$$\partial_1 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{1}$$
4) Next, we find the invertible matrices P_k, Q_k and diagonal matrix S_k so that the product $P_k \partial_k Q_k = S_k$. We also do the same for $k+1$. This representation is the **Smith Normal Form** of the matrix. Computing the Smith Normal form of a matrix employs different linear algebra operations, such as pivot calculation, swapping rows and columns of a matrix etc [1].
   In **Tri-Loop** the Smith Normal Form for ∂_1 is $P_1 \partial_1 Q_1 = S_1$:
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

[1] A nice description of the procedure may be found here: http://sierra.nmsu.edu/morandi/notes/SmithNormalForm.pdf

```
proc matmultmod (M, N, mod = 2) {
  var C : [M.domain.dim(1), N.domain.dim(2)] int;
  forall (i,j) in C.domain {
    C[i,j] = (+ reduce (M[i, M.domain.dim(2)]
     * N[M.domain.dim(2), j])) % mod;
  return C; }
proc swap_rows(i,j,M) {
  var N = M;
  N[i, ..] <=> N[j, ..];
  return N; }
proc swap_columns(i,j,M) {
 var N = M;
 N[.., i] <=> N[.., j];
 return N;}
proc add_to_row(M,i,j,ri=1,rj=1) {
  var N = M;
  N[i, ..]  = (ri * N[i, ..] + rj * N[j, ..]) % 2;
  return N; }
proc add_to_column(M,i,j,ci=1,cj=1) {
 var N = M;
 N[.., i]  = (ci * N[.., i] + cj * N[..,j]) % 2;
 return N;}
proc calculateRank(M) return + reduce
  [i in M.domain.dim(2)] (max reduce M[.., i]);
```

Listing 1: Linear algebra operations in $\mathbb{Z}_2$ in Chapel

and for ∂_2 is $P_2 \partial_2 Q_2 = S_2$:
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3}$$

5) Then, the Betti number for $k = 1$ is calculated as: number of columns of (S_1) − rank(S_1) − rank(S_2) = $5 - 3 - 1 = 1$.

III. Chapel Implementation

We have implemented the Betti number computation in the Chapel Hypergraph Library (CHGL). CHGL provides methods for constructing distributed hypergraphs and includes a lightweight runtime on top of Chapel to support efficient execution of algorithms for irregular applications. The runtime features include message aggregation using the Chapel Aggregation Library [3], a distributed work queue, termination detection algorithms, etc. A hypergraph in CHGL is represented as a data structure that consists of two Chapel arrays for vertices and edges. Each of these arrays points to the arrays for every vertex and edge inclusions. Every inclusion is stored both at the included vertex and at the including edge.

The homology computation, in particular, the Betti number calculation involves different array-centric and linear algebra operations, including row and column interchange, pivot calculation, addition of rows and columns, rank computation, multiplication, and slicing. These operations, implemented in Chapel (listing 1), are not only concise, but also more intuitive than in Python (listing 2). While Chapel's linear algebra library [4] offers various operations on matrices (such as singular value decomposition, LU decomposition etc), our operations are in the $\mathbb{Z}_2$ field, so we wrote our own. Note,

```
def matmultmod(M,N,mod=2):
  return np.mod(np.matmul(M,N),mod)
def swap_rows(i,j,M):
  N = copy.deepcopy(M)
  N[i] = M[j]
  N[j] = M[i]
  return N
def swap_columns(i,j,M):
  N = swap_rows(i,j,M.transpose())
  return N.transpose()
def add_to_row(M,i,j,ri=1,rj=1,mod=2):
  N = copy.deepcopy(M)
  N[i] = np.mod(ri*N[i] + rj*N[j],[mod])
  return N
def add_to_column(M,i,j,ci=1,cj=1,mod=2):
  N = M.transpose()
  return
   add_to_row(N,i,j,ci,cj,mod=mod).transpose()
def calculaterank(M):
  return np.sum(M)
```

Listing 2: Linear algebra operations in $\mathbb{Z}_2$ in Python/Numpy

```
var cellSets : [0..#numLocales,
                0..#here.maxTaskPar]
                set(Cell);
var taskIdCounts : [0..#numLocales] atomic int;
forall e in hypergraph.getEdges()
  with (var tid : int =
   taskIdCounts[here.id].fetchAdd(1)) {
  var vertices = hypergraph.incidence(e);
  ref tmp = vertices[1..#vertices.size];
  var verticesInEdge :
   [1..#vertices.size] int = tmp.id;
  processCell(
   new Cell(verticesInEdge),
         cellSets[here.id, tid]);
}
```

Listing 3: Computation of the ASCs

for example, how the Chapel code for matrix multiplication is not only extremely concise, but also naturally parallel and distributed (Listing 1).

To implement the algorithm for computing the Betti numbers, the first step involves computing the Abstract Simplicial complexes (ASCs) for each hyperedge (i.e. powerset of each hyperedge) (listing 3). To do so, we maintain one set per task per locale to gather the ASCs. Each hyperedge computes the ASCs in parallel (Line 5).

Next we combine the k-cells generated on each locale. We employ Chapel's hashed distribution to map the associative domain of k-cells to a set of target locales (listing 4).

Once we construct the associative domains of k-cells, we use this domain as keys of a thread-safe map, allocated on locale 0 to group the k-cells based on their sizes (listing 5). Next we

```
var cellSet : domain(Cell, parSafe=true)
  dmapped Hashed(idxType=Cell);
for cset in cellSets {
  forall cell in cset with (ref cellSet) {
    cellSet += cell;
}}
```

Listing 4: Mapping associated domain of cells to locales

```
var kCellMap = new map(int,
  list(Cell, parSafe=true),parSafe=true);
forall cell in cellSet {
  kCellMap[cell.size - 1].append(cell);
}
```

Listing 5: Map to combine and group the k-cells

```
forall (_kCellsArray, kCellKey)
  in zip(kCellsArrayMap, kCellKeys){
  _kCellsArray = new owned
  kCellsArray(kCellMap[kCellKey].size);
  _kCellsArray.A = kCellMap[kCellKey].toArray();
  sort(_kCellsArray.A, comparator=absComparator);
}
```

Listing 6: Sorting each bin of k-cells

perform lexicographical sort of each of the bins containing the k-cells in parallel (listing 6). For this, we provide a customized comparator. The `forall` loop employs Chapel's zippered iteration technique.

We code the boundary matrices as 2D block-distributed arrays in Chapel (listing 7). Computing the Smith Normal Form involves the array-centric and linear algebra operations listed in listing 1.

IV. Experimental Results

We have implemented the Betti number computation in the Chapel hypergraph Library v0.3 [5]. Our code was compiled with Chapel version 1.20, with `--fast` flag to enable all compiler optimizations. We ran the experiments on one of the compute nodes of an Infiniband cluster, each equipped with a 20-core Intel Xeon processor and 132GB memory. All cores were involved in the experiments. We compare our implementation with two other homology packages: Perseus [6] (written in C++) and Eirene [7] (written in Julia). We report our results in Figure 2.

The original Smith Normal Form (legend CHGL in fig. 2) calculation in CHGL maintains a list of matrix transformations to compute the matrix inverse. These transformations are multiplied pairwise to find the invertible matrices. In this method, the number of transformations can vary widely across different iterations and hence the number of matrix multiplications varies too. As can be seen from Figure 2, this implementation is not very scalable, as the number of simplices increases. With the help of Chapel's visual profiler, we have confirmed that, indeed matrix multiplication is the most compute-intensive kernel in our implementation. To reduce this bottleneck, we

```
class Matrix {
 ..
 var D = {1..N, 1..M}
  dmapped Block(boundingBox = {1..N, 1..M});
 var matrix : [D] int;
 ..
}
```

Listing 7: Boundary matrix representation in Chapel

```
proc add_to_row(M, i, j,  mod = 2) {
  var N = M;
  N[i, ..]  = (N[i, ..] ^  N[j, ..]);
  return N;
}
```

Listing 8: Boolean equivalent of add_to_row operation

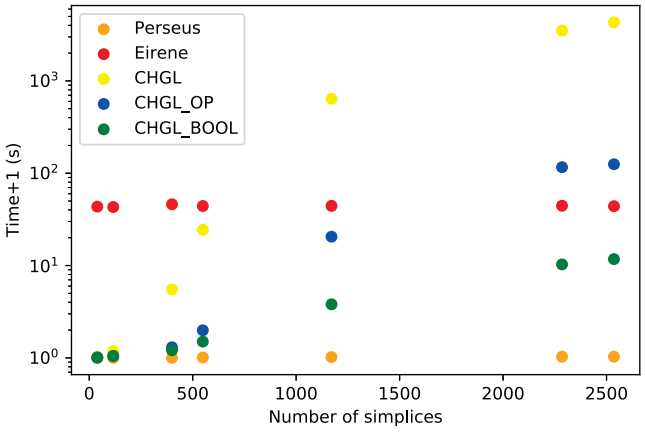

Fig. 2. Execution time of CHGL, Eirene and Perseus for Betti number calculation with $k = 1$ and $k = 2$.

have incorporated various optimizations in our code to improve the performance of the Betti number calculation. First, we have reformulated the calculation of Smith Normal Form to eliminate the requirement of performing explicit matrix-multiplications. Instead, in-place modification of the matrices are made on-the-fly to find the invertible matrices. With this optimization, the performance improves significantly (legend CHGL_OP in fig. 2). As a second optimization, since all of our computations are done in the $\mathbb{Z}_2$ field, we opted for boolean datatype and operations for boundary matrices instead of integer matrices (legend CHGL_BOOL in fig. 2). For example, the `add_to_row` operation for a matrix in the $\mathbb{Z}_2$ field translates to executing elementwise XOR operation (listing 8). Boolean representation of matrices and removing the requirement of performing matrix multiplications of a list of matrices delivers the best overall performance.

V. Conclusion and Future Work

In this paper, we discussed our experience with Chapel for the Betti number calculation when a dataset is represented as a hypergraph. Chapel's succinct syntax and various parallel constructs reduce the effort required for implementing complex, parallel algorithms involving various linear algebra operations. In terms of performance, Chapel-based implementation of the Betti number calculation out-performs the Julia-based implementation. As a future work, we are planning to work with the Chapel development team to understand the reasons for the performance gap between Chapel-based implementation and the C++-based implementation and improve the overall performance of the Chapel code.

References

[1] S. G. Aksoy, C. A. Joslyn, C. O. Marrero, B. Praggastis, and E. A. Purvine, "Hypernetwork science via high-order hypergraph walks," 2019, submitted. [Online]. Available: https://arxiv.org/abs/1906.11295

[2] L. Jenkins, T. Bhuiyan, S. Harun, C. Lightsey, D. Mentgen, S. Aksoy, T. Stavcnger, M. Zalewski, H. Medal, and C. Joslyn, "Chapel Hypergraph Library (CHGL)," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–6.

[3] L. Jenkins, M. Zalewski, and M. Ferguson, "Chapel Aggregation Library (CAL)," in *2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, Nov. 2018, pp. 34–43.

[4] "Linear algebra library in chapel," https://chapel-lang.org/docs/master/modules/packages/LinearAlgebra.html, 2020, [Online; accessed 2020].

[5] "Chapel hypergraph library (CHGL)," https://github.com/pnnl/chgl, [Online; accessed 2020].

[6] V. Nanda, "Perseus, the persistent homology software." http://www.sas.upenn.edu/ vnanda/perseus, [Online; accessed 2020].

[7] G. Henselman and R. Ghrist, "Matroid Filtrations and Computational Persistent Homology," *ArXiv e-prints*, Jun. 2016.